A Tour of the Dart Language

This page shows you how to use each major Dart feature, from variables and operators to classes and libraries, with the assumption that you already know how to program in another language.

To learn more about Dart's core libraries, see A Tour of the Dart Libraries.

Tip: You can play with most of these features using DartPad (learn more).

Open DartPad

Consult the Dart Language Specification whenever you want more details about a language feature.

A basic Dart program

The following code uses many of Dart's most basic features:

```
// Define a function.
printNumber(num aNumber) {
  print('The number is $aNumber.'); // Print to console.
}

// This is where the app starts executing.
main() {
  var number = 42; // Declare and initialize a variable.
  printNumber(number); // Call a function.
}
```

Here's what this program uses that applies to all (or almost all) Dart apps:

`// This is a comment.`
Use // to indicate that the rest of the line is a comment. Alternatively, use /* … */. For details, see Comments.

`num`
A type. Some of the other built-in types are String, int, and bool.

`42`
A number literal. Number literals are a kind of compile-time constant.

`print()`
A handy way to display output.

`'...'` (or `"..."`)
A string literal.

`$variableName` (or `${expression}`)
String interpolation: including a variable or expression's string equivalent inside of a string literal. For more information, see Strings.

`main()`
The special, *required*, top-level function where app execution starts. For more information, see The main() function.

`var`
A way to declare a variable without specifying its type.

> **Note:** Our code follows the conventions in the Dart style guide. For example, we use two-space indentation.

Important concepts

As you learn about the Dart language, keep these facts and concepts in mind:

- Everything you can place in a variable is an *object*, and every object is an instance of a *class*. Even numbers, functions, and `null` are objects. All objects inherit from the Object class.
- Specifying static types (such as `num` in the preceding example) clarifies your intent and enables static checking by tools. (You might notice when you're debugging your code that variables with no specified type get a special type: `dynamic`.) Specifying static types is optional in Dart 1.x, however Dart is moving towards being a fully type safe language.
- In strong mode, static and runtime checks ensure that your code is type safe, helping you catch bugs in development, rather than at runtime. Strong mode is optional in Dart 1.x, but not optional in Dart 2.
- Dart parses all your code before running it. You can provide tips to Dart—for example, by using types or compile-time constants—to catch errors or help your code run faster.
- Dart supports top-level functions (such as `main()`), as well as functions tied to a class or object (*static* and *instance methods*, respectively). You can also create functions within functions (*nested* or *local functions*).
- Similarly, Dart supports top-level *variables*, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as fields or properties.

- Unlike Java, Dart doesn't have the keywords `public`, `protected`, and `private`. If an identifier starts with an underscore (_), it's private to its library. For details, see Libraries and visibility.
- *Identifiers* can start with a letter or _, followed by any combination of those characters plus digits.
- Sometimes it matters whether something is an *expression* or a *statement*, so we'll be precise about those two words.
- Dart tools can report two kinds of problems: warnings and errors. Warnings are just indications that your code might not work, but they don't prevent your program from executing. Errors can be either compile-time or run-time. A compile-time error prevents the code from executing at all; a run-time error results in an exception being raised while the code executes.
- Dart 1.x has two *runtime modes*: production and checked. We recommend that you develop and debug in checked mode, and deploy to production mode. *Production mode* is the default runtime mode of a Dart program, optimized for speed. Production mode ignores assert statements and static types. *Checked mode* is a developer-friendly mode that helps you catch some type errors during runtime. For example, if you assign a non-number to a variable declared as a `num`, then checked mode throws an exception.

> **Dart 2 note**: Checked mode won't be in Dart 2. For more information, see Dart 2 Updates.

Keywords

The following table lists the words that the Dart language treats specially.

abstract [1]	deferred [1]	if	super
as [1]	do	implements [1]	switch
assert	dynamic [1]	import [1]	sync* [2]
async [2]	else	in	this
async* [2]	enum	is	throw
await [2]	export [1]	library [1]	true
break	external [1]	new	try
case	extends	null	typedef [1]
catch	factory [1]	operator [1]	var
class	false	part [1]	void

const	final	rethrow	while
continue	finally	return	with
covariant [1]	for	set [1]	yield [2]
default	get [1]	static [1]	yield* [2]

[1] Words with the superscript **1** are **built-in identifiers**. Avoid using built-in identifiers as identifiers. A compile-time error happens if you try to use a built-in identifier for a class or type name.

[2] Words with the superscript **2** are newer, limited reserved words related to **asynchrony** support added after Dart's 1.0 release. You can't use `async`, `await`, or `yield` as an identifier in any function body marked with `async`, `async*`, or `sync*`. For more information, see Asynchrony support.

All other words in the keyword table are **reserved words**. You can't use reserved words as identifiers.

Variables

Here's an example of creating a variable and assigning a value to it:

```
var name = 'Bob';
```

Variables are references. The variable called name contains a reference to a String object with a value of "Bob".

Default value

Uninitialized variables have an initial value of `null`. Even variables with numeric types are initially null, because numbers are objects.

```
int lineCount;
assert(lineCount == null);
// Variables (even if they will be numbers) are initially null.
```

Note: The `assert()` call is ignored in production mode. In checked mode, `assert(`*`condition`*`)` throws an exception unless *condition* is true. For details, see the Assert section.

> **Dart 2 note**: Checked mode won't be in Dart 2. For more information, see Dart 2 Updates.

Optional types

> **Dart 2 note**: Types won't be optional in Dart 2, but you'll still be able to omit some type annotations, thanks to type inference. For more information, see Dart 2 Updates.

You have the option of adding static types to your variable declarations:

```
String name = 'Bob';
```

Adding types is a way to clearly express your intent. Tools such as compilers and editors can use these types to help you, by providing code completion and early warnings for bugs and code completion.

> **Note:** This page follows the style guide recommendation of using `var`, rather than type annotations, for local variables. Even under strong mode, you can specify `var`—the analyzer infers the type where possible.

Final and const

If you never intend to change a variable, use `final` or `const`, either instead of `var` or in addition to a type. A final variable can be set only once; a const variable is a compile-time constant. (Const variables are implicitly final.) A final top-level or class variable is initialized the first time it's used.

> **Note:** Instance variables can be `final` but not `const`.

Here's an example of creating and setting a final variable:

```
final name = 'Bob'; // Without a type annotation
// name = 'Alice';  // Uncommenting this causes an error
final String nickname = 'Bobby';
```

Use `const` for variables that you want to be compile-time constants. If the const variable is at the class level, mark it `static const`. Where you declare the variable, set the value to a compile-time constant such as a number or string literal, a const variable, or the result of an arithmetic operation on constant numbers:

```
const bar = 1000000; // Unit of pressure (dynes/cm2)
const double atm = 1.01325 * bar; // Standard atmosphere
```

The `const` keyword isn't just for declaring constant variables. You can also use it to create constant *values*, as well as to declare constructors that *create* constant values. Any variable can have a constant value.

```
// Note: [] creates an empty list.
// const [] creates an empty, immutable list (EIL).
var foo = const []; // foo is currently an EIL.
final bar = const []; // bar will always be an EIL.
const baz = const []; // baz is a compile-time constant EIL.

// You can change the value of a non-final, non-const variable,
// even if it used to have a const value.
foo = [];

// You can't change the value of a final or const variable.
// bar = []; // Unhandled exception.
// baz = []; // Unhandled exception.
```

For more information on using `const` to create constant values, see Lists, Maps, and Classes.

Built-in types

The Dart language has special support for the following types:

- numbers
- strings
- booleans
- lists (also known as *arrays*)
- maps
- runes (for expressing Unicode characters in a string)
- symbols

You can initialize an object of any of these special types using a literal. For example, `'this is a string'` is a string literal, and `true` is a boolean literal.

Because every variable in Dart refers to an object—an instance of a *class*—you can usually use *constructors* to initialize variables. Some of the built-in types have their own constructors. For example, you can use the `Map()` constructor to create a map, using code such as `new Map()`.

Numbers

Dart numbers come in two flavors:

int
Integer values, which generally should be in the range -2^{53} to 2^{53}

double
64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard

Both `int` and `double` are subtypes of num. The num type includes basic operators such as +, -, /, and *, and is also where you'll find `abs()`, `ceil()`, and `floor()`, among other methods. (Bitwise operators, such as >>, are defined in the `int` class.) If num and its subtypes don't have what you're looking for, the dart:math library might.

> **Warning:** Integers outside of the -2^{53} to 2^{53} range currently behave differently in JavaScript produced from Dart code than they do when the same Dart code runs in the Dart VM. The reason is that Dart is specified to have arbitrary-precision integers, but JavaScript isn't. See issue 1533 for details.

Integers are numbers without a decimal point. Here are some examples of defining integer literals:

```
int x = 1;
int hex = 0xDEADBEEF;
int bigInt = 346534658346524376592384765923749587398457294759347029438709349347;
```

If a number includes a decimal, it is a double. Here are some examples of defining double literals:

```
double y = 1.1;
double exponents = 1.42e5;
```

Here's how you turn a string into a number, or vice versa:

```
// String -> int
var one = int.parse('1');
assert(one == 1);

// String -> double
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);

// int -> String
String oneAsString = 1.toString();
assert(oneAsString == '1');

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

The int type specifies the traditional bitwise shift (<<, >>), AND (&), and OR (|) operators. For example:

```
assert((3 << 1) == 6); // 0011 << 1 == 0110
assert((3 >> 1) == 1); // 0011 >> 1 == 0001
assert((3 | 4) == 7); // 0011 | 0100 == 0111
```

Literal numbers are compile-time constants. Many arithmetic expressions are also compile-time constants, as long as their operands are compile-time constants that evaluate to numbers.

```
const msPerSecond = 1000;
const secondsUntilRetry = 5;
const msUntilRetry = secondsUntilRetry * msPerSecond;
```

Strings

A Dart string is a sequence of UTF-16 code units. You can use either single or double quotes to create a string:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

You can put the value of an expression inside a string by using `${expression}`. If the expression is an identifier, you can skip the {}. To get the string corresponding to an object, Dart calls the object's `toString()` method.

```dart
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
    'Dart has string interpolation, ' +
        'which is very handy.');
assert('That deserves all caps. ' +
        '${s.toUpperCase()} is very handy!' ==
    'That deserves all caps. ' +
        'STRING INTERPOLATION is very handy!');
```

Note: The == operator tests whether two objects are equivalent. Two strings are equivalent if they contain the same sequence of code units.

You can concatenate strings using adjacent string literals or the + operator:

```dart
var s1 = 'String '
    'concatenation'
    " works even over line breaks.";
assert(s1 ==
    'String concatenation works even over '
    'line breaks.');

var s2 = 'The + operator ' + 'works, as well.';
assert(s2 == 'The + operator works, as well.');
```

Another way to create a multi-line string: use a triple quote with either single or double quotation marks:

```dart
var s1 = '''
You can create
multi-line strings like this one.
''';

var s2 = """This is also a
multi-line string.""";
```

You can create a “raw” string by prefixing it with r:

```dart
var s = r"In a raw string, even \n isn't special.";
```

See Runes for details on how to express Unicode characters in a string.

Literal strings are compile-time constants, as long as any interpolated expression is a compile-time constant that evaluates to null or a numeric, string, or boolean value.

```
// These work in a const string.
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// These do NOT work in a const string.
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = const [1, 2, 3];

const validConstString = '$aConstNum $aConstBool $aConstString';
// const invalidConstString = '$aNum $aBool $aString $aConstList';
```

For more information on using strings, see Strings and regular expressions.

Booleans

To represent boolean values, Dart has a type named `bool`. Only two objects have type bool: the boolean literals `true` and `false`, which are both compile-time constants.

When Dart expects a boolean value, only the value `true` is treated as true. All other values are treated as false. Unlike in JavaScript, values such as `1`, `"aString"`, and `someObject` are all treated as false.

For example, consider the following code, which is valid both as JavaScript and as Dart code:

```
var name = 'Bob';
if (name) {
  // Prints in JavaScript, not in Dart.
  print('You have a name!');
}
```

If you run this code as JavaScript, it prints "You have a name!" because `name` is a non-null object. However, in Dart running in *production mode*, the preceding code doesn't print at all because `name` is converted to `false` (because `name != true`). In Dart running in *checked mode*, the preceding code throws an exception because the `name` variable is not a bool.

Dart 2 note: Checked mode won't be in Dart 2. For more information, see Dart 2 Updates.

Here's another example of code that behaves differently in JavaScript and Dart:

```
if (1) {
  print('JS prints this line.');
} else {
  print('Dart in production mode prints this line.');
  // However, in checked mode, if (1) throws an
  // exception because 1 is not boolean.
}
```

Note: The previous two samples work only in production mode, not checked mode. In checked mode, an exception is thrown if a non-boolean is used when a boolean value is expected.

Dart's treatment of booleans is designed to avoid the strange behaviors that can arise when many values can be treated as true. What this means for you is that, instead of using code like `if (`*`nonbooleanValue`*`)`, you should instead explicitly check for values. For example:

```
// Check for an empty string.
var fullName = '';
assert(fullName.isEmpty);

// Check for zero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Check for null.
var unicorn;
assert(unicorn == null);

// Check for NaN.
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

Lists

Perhaps the most common collection in nearly every programming language is the *array*, or ordered group of objects. In Dart, arrays are List objects, so we usually just call them *lists*.

Dart list literals look like JavaScript array literals. Here's a simple Dart list:

```
var list = [1, 2, 3];
```

> **Note:** If you’re wondering whether this list literal is strong mode compliant, it is. Thanks to type inference, the analyzer infers that `list` has type `List<int>`. If you try to add non-integer objects to this list under strong mode, the analyzer raises an error.

Lists use zero-based indexing, where 0 is the index of the first element and `list.length - 1` is the index of the last element. You can get a list’s length and refer to list elements just as you would in JavaScript:

```
var list = [1, 2, 3];
assert(list.length == 3);
assert(list[1] == 2);

list[1] = 1;
assert(list[1] == 1);
```

To create a list that’s a compile-time constant, add `const` before the list literal:

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // Uncommenting this causes an error.
```

The List type has many handy methods for manipulating lists. For more information about lists, see Generics and Collections.

Maps

In general, a map is an object that associates keys and values. Both keys and values can be any type of object. Each *key* occurs only once, but you can use the same *value* multiple times. Dart support for maps is provided by map literals and the Map type.

Here are a couple of simple Dart maps, created using map literals:

```
var gifts = {
  // Key:    Value
  'first': 'partridge',
  'second': 'turtledoves',
  'fifth': 'golden rings'
};

var nobleGases = {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};
```

> **Note:** If you're wondering whether these map literals are strong mode compliant, they are. Thanks to type inference, the analyzer infers that `gifts`, for example, has the type `Map<String, String>`. If you try to add non-string objects to this map under strong mode, the analyzer raises an error.

You can create the same objects using a Map constructor:

```
var gifts = new Map();
gifts['first'] = 'partridge';
gifts['second'] = 'turtledoves';
gifts['fifth'] = 'golden rings';

var nobleGases = new Map();
nobleGases[2] = 'helium';
nobleGases[10] = 'neon';
nobleGases[18] = 'argon';
```

Add a new key-value pair to an existing map just as you would in JavaScript:

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // Add a key-value pair
```

Retrieve a value from a map the same way you would in JavaScript:

```
var gifts = {'first': 'partridge'};
assert(gifts['first'] == 'partridge');
```

If you look for a key that isn't in a map, you get a null in return:

```
var gifts = {'first': 'partridge'};
assert(gifts['fifth'] == null);
```

Use `.length` to get the number of key-value pairs in the map:

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds';
assert(gifts.length == 2);
```

To create a map that's a compile-time constant, add `const` before the map literal:

```
final constantMap = const {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};

// constantMap[2] = 'Helium'; // Uncommenting this causes an error.
```

For more information about maps, see Generics and Maps.

Runes

In Dart, runes are the UTF-32 code points of a string.

Unicode defines a unique numeric value for each letter, digit, and symbol used in all of the world's writing systems. Because a Dart string is a sequence of UTF-16 code units, expressing 32-bit Unicode values within a string requires special syntax.

The usual way to express a Unicode code point is `\uXXXX`, where XXXX is a 4-digit hexidecimal value. For example, the heart character (♥) is `\u2665`. To specify more or less than 4 hex digits, place the value in curly brackets. For example, the laughing emoji (😆) is `\u{1f600}`.

The String class has several properties you can use to extract rune information. The `codeUnitAt` and `codeUnit` properties return 16-bit code units. Use the `runes` property to get the runes of a string.

The following example illustrates the relationship between runes, 16-bit code units, and 32-bit code points. Click the run button (▶) to see runes in action.

xxxxxxxxxx

Console output

☑ Strong mode (?)

```
main() {
```

> **Note:** Be careful when manipulating runes using list operations. This approach can easily break down, depending on the particular language, character set, and operation. For more information, see How do I reverse a String in Dart? on Stack Overflow.

Symbols

A Symbol object represents an operator or identifier declared in a Dart program. You might never need to use symbols, but they're invaluable for APIs that refer to identifiers by name, because minification changes identifier names but not identifier symbols.

To get the symbol for an identifier, use a symbol literal, which is just # followed by the identifier:

```
#radix
#bar
```

Symbol literals are compile-time constants.

Functions

Dart is a true object-oriented language, so even functions are objects and have a type, Function. This means that functions can be assigned to variables or passed as arguments to other functions. You can also call an instance of a Dart class as if it were a function. For details, see Callable classes.

Here's an example of implementing a function:

```
bool isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

Although Effective Dart recommends type annotations for public APIs, the function still works if you omit the types:

```
isNoble(atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

For functions that contain just one expression, you can use a shorthand syntax:

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

The => *expr* syntax is a shorthand for `{ return` *`expr`*`; }`. The => notation is sometimes referred to as *fat arrow* syntax.

Note: Only an *expression*—not a *statement*—can appear between the arrow (=>) and the semicolon (;). For example, you can't put an if statement there, but you can use a conditional expression.

A function can have two types of parameters: required and optional. The required parameters are listed first, followed by any optional parameters.

Optional parameters

Optional parameters can be either positional or named, but not both.

Optional named parameters

When calling a function, you can specify named parameters using `paramName: value`. For example:

```
enableFlags(bold: true, hidden: false);
```

When defining a function, use `{param1, param2, …}` to specify named parameters:

```
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool bold, bool hidden}) {
  // ...
}
```

Optional positional parameters

Wrapping a set of function parameters in `[]` marks them as optional positional parameters:

```
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Here's an example of calling this function without the optional parameter:

```
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
```

And here's an example of calling this function with the third parameter:

```
assert(say('Bob', 'Howdy', 'smoke signal') ==
    'Bob says Howdy with a smoke signal');
```

Default parameter values

Your function can use = to define default values for both named and positional parameters. The default values must be compile-time constants. If no default value is provided, the default value is `null`.

Here's an example of setting default values for named parameters:

```
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool bold = false, bool hidden = false}) {
  // ...
}

// bold will be true; hidden will be false.
enableFlags(bold: true);
```

> **Version note:** Old code might use a colon (`:`) instead of = to set default values of named parameters. The reason is that before SDK 1.21, only `:` was supported for named parameters. That support is likely to be deprecated, so we recommend that you **use = to specify default values, and specify an SDK version of 1.21 or higher.**

The next example shows how to set default values for positional parameters:

```
String say(String from, String msg,
    [String device = 'carrier pigeon', String mood]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  if (mood != null) {
    result = '$result (in a $mood mood)';
  }
  return result;
}

assert(say('Bob', 'Howdy') ==
    'Bob says Howdy with a carrier pigeon');
```

You can also pass lists or maps as default values. The following example defines a function, `doStuff()`, that specifies a default list for the `list` parameter and a default map for the `gifts` parameter.

```
void doStuff(
    {List<int> list = const [1, 2, 3],
    Map<String, String> gifts = const {
      'first': 'paper',
      'second': 'cotton',
      'third': 'leather'
    }}) {
  print('list:  $list');
  print('gifts: $gifts');
}
```

The main() function

Every app must have a top-level `main()` function, which serves as the entrypoint to the app. The `main()` function returns `void` and has an optional `List<String>` parameter for arguments.

Here's an example of the `main()` function for a web app:

```
void main() {
  querySelector('#sample_text_id')
    ..text = 'Click me!'
    ..onClick.listen(reverseText);
}
```

> **Note:** The `..` syntax in the preceding code is called a cascade. With cascades, you can perform multiple operations on the members of a single object.

Here's an example of the `main()` function for a command-line app that takes arguments:

```
// Run the app like this: dart args.dart 1 test
void main(List<String> arguments) {
  print(arguments);

  assert(arguments.length == 2);
  assert(int.parse(arguments[0]) == 1);
  assert(arguments[1] == 'test');
}
```

You can use the args library to define and parse command-line arguments.

Functions as first-class objects

You can pass a function as a parameter to another function. For example:

```
void printElement(int element) {
  print(element);
}

var list = [1, 2, 3];

// Pass printElement as a parameter.
list.forEach(printElement);
```

You can also assign a function to a variable, such as:

```
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(loudify('hello') == '!!! HELLO !!!');
```

This example uses an anonymous function. More about those in the next section.

Anonymous functions

Most functions are named, such as `main()` or `printElement()`. You can also create a nameless function called an *anonymous function*, or sometimes a *lambda* or *closure*. You might assign an anonymous function to a variable so that, for example, you can add or remove it from a collection.

An anonymous function looks similar to a named function— zero or more parameters, separated by commas and optionally typed, between parentheses.

> **Dart 2 note**: Types won't be optional in Dart 2, but you'll still be able to omit some type annotations, thanks to type inference. For more information, see Dart 2 Updates.

The code block that follows contains the function's body:

```
([[Type] param1[, …]]) {
  codeBlock;
};
```

The following example defines an anonymous function with an untyped parameter, `item`. The function, invoked for each item in the list, prints a string that includes the value at the specified index.

```
var list = ['apples', 'bananas', 'oranges'];
list.forEach((item) {
  print('${list.indexOf(item)}: $item');
});
```

Click the run button ( ) to execute the code.

If the function contains only one statement, you can shorten it using fat arrow notation. Paste the following line into DartPad and click run to verify that it is functionally equivalent.

```
list.forEach(
    (item) => print('${list.indexOf(item)}: $item'));
```

Lexical scope

Dart is a lexically scoped language, which means that the scope of variables is determined statically, simply by the layout of the code. You can “follow the curly braces outwards” to see if a variable is in scope.

Here is an example of nested functions with variables at each scope level:

```
bool topLevel = true;

void main() {
  var insideMain = true;

  void myFunction() {
    var insideFunction = true;

    void nestedFunction() {
      var insideNestedFunction = true;

      assert(topLevel);
      assert(insideMain);
      assert(insideFunction);
      assert(insideNestedFunction);
    }
  }
}
```

Notice how `nestedFunction()` can use variables from every level, all the way up to the top level.

Lexical closures

A *closure* is a function object that has access to variables in its lexical scope, even when the function is used outside of its original scope.

Functions can close over variables defined in surrounding scopes. In the following example, `makeAdder()` captures the variable addBy. Wherever the returned function goes, it remembers `addBy`.

```
/// Returns a function that adds [addBy] to the
/// function's argument.
Function makeAdder(num addBy) {
  return (num i) => addBy + i;
}

void main() {
  // Create a function that adds 2.
  var add2 = makeAdder(2);

  // Create a function that adds 4.
  var add4 = makeAdder(4);

  assert(add2(3) == 5);
  assert(add4(3) == 7);
}
```

Testing functions for equality

Here’s an example of testing top-level functions, static methods, and instance methods for equality:

```dart
void foo() {} // A top-level function

class A {
  static void bar() {} // A static method
  void baz() {} // An instance method
}

void main() {
  var x;

  // Comparing top-level functions.
  x = foo;
  assert(foo == x);

  // Comparing static methods.
  x = A.bar;
  assert(A.bar == x);

  // Comparing instance methods.
  var v = new A(); // Instance #1 of A
  var w = new A(); // Instance #2 of A
  var y = w;
  x = w.baz;

  // These closures refer to the same instance (#2),
  // so they're equal.
  assert(y.baz == x);

  // These closures refer to different instances,
  // so they're unequal.
  assert(v.baz != w.baz);
}
```

Return values

All functions return a value. If no return value is specified, the statement `return null;` is implicitly appended to the function body.

```dart
foo() {}

assert(foo() == null);
```

Operators

Dart defines the operators shown in the following table. You can override many of these operators, as described in Overridable operators.

Description	Operator
unary postfix	*`expr`*`++` *`expr`*`--` `()` `[]` `.` `?.`
unary prefix	`-`*`expr`* `!`*`expr`* `~`*`expr`* `++`*`expr`* `--`*`expr`*
multiplicative	`*` `/` `%` `~/`
additive	`+` `-`
shift	`<<` `>>`
bitwise AND	`&`
bitwise XOR	`^`
bitwise OR	`|`
relational and type test	`>=` `>` `<=` `<` `as` `is` `is!`
equality	`==` `!=`
logical AND	`&&`
logical OR	`||`
if null	`??`
conditional	*`expr1`* `?` *`expr2`* `:` *`expr3`*
cascade	`..`
assignment	`=` `*=` `/=` `~/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=` `??=`

When you use operators, you create expressions. Here are some examples of operator expressions:

```
a++
a + b
a = b
a == b
c ? a : b
a is T
```

In the operator table, each operator has higher precedence than the operators in the rows that follow it. For example, the multiplicative operator `%` has higher precedence than (and thus executes before) the equality operator `==`, which has higher precedence than the logical AND operator &&. That precedence means that the following two lines of code execute the same way:

```
// Parentheses improve readability.
if ((n % i == 0) && (d % i == 0)) ...

// Harder to read, but equivalent.
if (n % i == 0 && d % i == 0) ...
```

Warning: For operators that work on two operands, the leftmost operand determines which version of the operator is used. For example, if you have a Vector object and a Point object, `aVector + aPoint` uses the Vector version of +.

Arithmetic operators

Dart supports the usual arithmetic operators, as shown in the following table.

Operator	Meaning
`+`	Add
`–`	Subtract
`-expr`	Unary minus, also known as negation (reverse the sign of the expression)
`*`	Multiply
`/`	Divide
`~/`	Divide, returning an integer result
`%`	Get the remainder of an integer division (modulo)

Example:

```
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // Result is a double
assert(5 ~/ 2 == 2); // Result is an int
assert(5 % 2 == 1); // Remainder

assert('5/2 = ${5~/2} r ${5%2}' == '5/2 = 2 r 1');
```

Dart also supports both prefix and postfix increment and decrement operators.

Operator	Meaning
++*var*	*var* = *var* + 1 (expression value is *var* + 1)
var++	*var* = *var* + 1 (expression value is *var*)
--*var*	*var* = *var* – 1 (expression value is *var* – 1)
var--	*var* = *var* – 1 (expression value is *var*)

Example:

```
var a, b;

a = 0;
b = ++a; // Increment a before b gets its value.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Increment a AFTER b gets its value.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Decrement a before b gets its value.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Decrement a AFTER b gets its value.
assert(a != b); // -1 != 0
```

Equality and relational operators

The following table lists the meanings of equality and relational operators.

Operator	Meaning
==	Equal; see discussion below
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

To test whether two objects x and y represent the same thing, use the == operator. (In the rare case where you need to know whether two objects are the exact same object, use the identical() function instead.) Here's how the == operator works:

1. If *x* or *y* is null, return true if both are null, and false if only one is null.
2. Return the result of the method invocation `x.==(y)`. (That's right, operators such as == are methods that are invoked on their first operand. You can even override many operators, including ==, as you'll see in Overridable operators.)

Here's an example of using each of the equality and relational operators:

```
assert(2 == 2);
assert(2 != 3);
assert(3 > 2);
assert(2 < 3);
assert(3 >= 3);
assert(2 <= 3);
```

Type test operators

The `as`, `is`, and `is!` operators are handy for checking types at runtime.

Operator	**Meaning**
`as`	Typecast
`is`	True if the object has the specified type
`is!`	False if the object has the specified type

The result of `obj is T` is true if `obj` implements the interface specified by T. For example, `obj is Object` is always true.

Use the `as` operator to cast an object to a particular type. In general, you should use it as a shorthand for an `is` test on an object following by an expression using that object. For example, consider the following code:

```
if (emp is Person) {
  // Type check
  emp.firstName = 'Bob';
}
```

You can make the code shorter using the as operator:

```
(emp as Person).firstName = 'Bob';
```

Note: The code isn't equivalent. If emp is null or not a Person, the first example (with `is`) does nothing; the second (with `as`) throws an exception.

Assignment operators

As you've already seen, you can assign values using the = operator. To assign only if the assigned-to variable is null, use the `??=` operator.

```
// Assign value to a
a = value;
// Assign value to b if b is null; otherwise, b stays the same
b ??= value;
```

Compound assignment operators such as += combine an operation with an assignment.

=	–=	/=	%=	>>=	^=
+=	*=	~/=	<<=	&=	|=

Here's how compound assignment operators work:

	Compound assignment	**Equivalent expression**
For an operator *op*:	`a op= b`	`a = a op b`
Example:	`a += b`	`a = a + b`

The following example uses assignment and compound assignment operators:

```
var a = 2; // Assign using =
a *= 3; // Assign and multiply: a = a * 3
assert(a == 6);
```

Logical operators

You can invert or combine boolean expressions using the logical operators.

Operator	**Meaning**
`!expr`	inverts the following expression (changes false to true, and vice versa)
`||`	logical OR

Operator	Meaning
&&	logical AND

Here’s an example of using the logical operators:

```
if (!done && (col == 0 || col == 3)) {
  // ...Do something...
}
```

Bitwise and shift operators

You can manipulate the individual bits of numbers in Dart. Usually, you’d use these bitwise and shift operators with integers.

Operator	Meaning
&	AND
|	OR
^	XOR
~expr	Unary bitwise complement (0s become 1s; 1s become 0s)
<<	Shift left
>>	Shift right

Here’s an example of using bitwise and shift operators:

```
final value = 0x22;
final bitmask = 0x0f;

assert((value & bitmask) == 0x02); // AND
assert((value & ~bitmask) == 0x20); // AND NOT
assert((value | bitmask) == 0x2f); // OR
assert((value ^ bitmask) == 0x2d); // XOR
assert((value << 4) == 0x220); // Shift left
assert((value >> 4) == 0x02); // Shift right
```

Conditional expressions

Dart has two operators that let you concisely evaluate expressions that might otherwise require if-else statements:

`condition ? expr1 : expr2`
If *condition* is true, evaluates *expr1* (and returns its value); otherwise, evaluates and returns the value of *expr2*.

`expr1 ?? expr2`
If *expr1* is non-null, returns its value; otherwise, evaluates and returns the value of *expr2*.

When you need to assign a value based on a boolean expression, consider using `?:`.

```
var visibility = isPublic ? 'public' : 'private';
```

If the boolean expression tests for null, consider using `??`.

```
String playerName(String name) => name ?? 'Guest';
```

The previous example could have been written at least two other ways, but not as succinctly:

```
// Slightly longer version uses ?: operator.
String playerName(String name) => name != null ? name : 'Guest';

// Very long version uses if-else statement.
String playerName(String name) {
  if (name != null) {
    return name;
  } else {
    return 'Guest';
  }
}
```

Cascade notation (..)

Cascades (`..`) allow you to make a sequence of operations on the same object. In addition to function calls, you can also access fields on that same object. This often saves you the step of creating a temporary variable and allows you to write more fluid code.

Consider the following code:

```
querySelector('#confirm') // Get an object.
  ..text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

The first method call, `querySelector()`, returns a selector object. The code that follows the cascade notation operates on this selector object, ignoring any subsequent values that might be returned.

The previous example is equivalent to:

```
var button = querySelector('#confirm');
button.text = 'Confirm';
button.classes.add('important');
button.onClick.listen((e) => window.alert('Confirmed!'));
```

You can also nest your cascades. For example:

```
final addressBook = (new AddressBookBuilder()
      ..name = 'jenny'
      ..email = 'jenny@example.com'
      ..phone = (new PhoneNumberBuilder()
            ..number = '415-555-0100'
            ..label = 'home')
          .build())
    .build();
```

Be careful to construct your cascade on a function that returns an actual object. For example, the following code fails:

```
var sb = new StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: method 'write' isn't defined for 'void'.
```

The `sb.write()` call returns void, and you can't construct a cascade on `void`.

> **Note:** Strictly speaking, the "double dot" notation for cascades is not an operator. It's just part of the Dart syntax.

Other operators

You've seen most of the remaining operators in other examples:

Operator	Name	Meaning
()	Function application	Represents a function call
[]	List access	Refers to the value at the specified index in the list

Operator	Name	Meaning
`.`	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
`?.`	Conditional member access	Like `.`, but the leftmost operand can be null; example: `foo?.bar` selects property `bar` from expression `foo` unless `foo` is null (in which case the value of `foo?.bar` is null)

For more information about the `.`, `?.`, and `..` operators, see Classes.

Control flow statements

You can control the flow of your Dart code using any of the following:

- `if` and `else`
- `for` loops
- `while` and do-`while` loops
- `break` and `continue`
- `switch` and `case`
- `assert`

You can also affect the control flow using `try-catch` and `throw`, as explained in Exceptions.

If and else

Dart supports `if` statements with optional `else` statements, as the next sample shows. Also see conditional expressions.

```
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

Remember, unlike JavaScript, Dart treats all values other than `true` as `false`. See Booleans for more information.

For loops

You can iterate with the standard `for` loop. For example:

```
var message = new StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
```

Closures inside of Dart's `for` loops capture the *value* of the index, avoiding a common pitfall found in JavaScript. For example, consider:

```
var callbacks = [];
for (var i = 0; i < 2; i++) {
  callbacks.add(() => print(i));
}
callbacks.forEach((c) => c());
```

The output is `0` and then `1`, as expected. In contrast, the example would print 2 and then 2 in JavaScript.

If the object that you are iterating over is an Iterable, you can use the forEach() method. Using `forEach()` is a good option if you don't need to know the current iteration counter:

```
candidates.forEach((candidate) => candidate.interview());
```

Iterable classes such as List and Set also support the `for-in` form of iteration:

```
var collection = [0, 1, 2];
for (var x in collection) {
  print(x); // 0 1 2
}
```

While and do-while

A `while` loop evaluates the condition before the loop:

```
while (!isDone()) {
  doSomething();
}
```

A `do-while` loop evaluates the condition *after* the loop:

```
do {
  printLine();
} while (!atEndOfPage());
```

Break and continue

Use `break` to stop looping:

```
while (true) {
  if (shutDownRequested()) break;
  processIncomingRequests();
}
```

Use `continue` to skip to the next loop iteration:

```
for (int i = 0; i < candidates.length; i++) {
  var candidate = candidates[i];
  if (candidate.yearsExperience < 5) {
    continue;
  }
  candidate.interview();
}
```

You might write that example differently if you're using an Iterable such as a list or set:

```
candidates
    .where((c) => c.yearsExperience >= 5)
    .forEach((c) => c.interview());
```

Switch and case

Switch statements in Dart compare integer, string, or compile-time constants using ==. The compared objects must all be instances of the same class (and not of any of its subtypes), and the class must not override ==. Enumerated types work well in `switch` statements.

> **Note:** Switch statements in Dart are intended for limited circumstances, such as in interpreters or scanners.

Each non-empty `case` clause ends with a `break` statement, as a rule. Other valid ways to end a non-empty `case` clause are a `continue`, `throw`, or `return` statement.

Use a `default` clause to execute code when no `case` clause matches:

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}
```

The following example omits the `break` statement in a `case` clause, thus generating an error:

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Missing break

  case 'CLOSED':
    executeClosed();
    break;
}
```

However, Dart does support empty `case` clauses, allowing a form of fall-through:

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED': // Empty case falls through.
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}
```

If you really want fall-through, you can use a `continue` statement and a label:

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED':
    executeClosed();
    continue nowClosed;
  // Continues executing at the nowClosed label.

  nowClosed:
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}
```

A `case` clause can have local variables, which are visible only inside the scope of that clause.

Assert

Use an `assert` statement to disrupt normal execution if a boolean condition is false. You can find examples of assert statements throughout this tour. Here are some more:

```
// Make sure the variable has a non-null value.
assert(text != null);

// Make sure the value is less than 100.
assert(number < 100);

// Make sure this is an https URL.
assert(urlString.startsWith('https'));
```

Note: Assert statements work only in checked mode. They have no effect in production mode.

Dart 2 note: Checked mode won't be in Dart 2. For more information, see Dart 2 Updates.

To attach a message to an assert, add a string as the second argument.

```
assert(urlString.startsWith('https'),
    'URL ($urlString) should start with "https".');
```

Version note: The second argument was introduced in SDK 1.22.

The first argument to `assert` can be any expression that resolves to a boolean value or to a function. If the expression's value or function's return value is true, the assertion succeeds and execution continues. If it's false, the assertion fails and an exception (an AssertionError) is thrown.

Exceptions

Your Dart code can throw and catch exceptions. Exceptions are errors indicating that something unexpected happened. If the exception isn't caught, the isolate that raised the exception is suspended, and typically the isolate and its program are terminated.

In contrast to Java, all of Dart's exceptions are unchecked exceptions. Methods do not declare which exceptions they might throw, and you are not required to catch any exceptions.

Dart provides Exception and Error types, as well as numerous predefined subtypes. You can, of course, define your own exceptions. However, Dart programs can throw any non-null object—not just Exception and Error objects—as an exception.

Throw

Here's an example of throwing, or *raising*, an exception:

```
throw new FormatException('Expected at least 1 section');
```

You can also throw arbitrary objects:

```
throw 'Out of llamas!';
```

Note: Production-quality code usually throws types that implement Error or Exception.

Because throwing an exception is an expression, you can throw exceptions in => statements, as well as anywhere else that allows expressions:

```
void distanceTo(Point other) =>
    throw new UnimplementedError();
```

Catch

Catching, or capturing, an exception stops the exception from propagating (unless you rethrow the exception). Catching an exception gives you a chance to handle it:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  buyMoreLlamas();
}
```

To handle code that can throw more than one type of exception, you can specify multiple catch clauses. The first catch clause that matches the thrown object's type handles the exception. If the catch clause does not specify a type, that clause can handle any type of thrown object:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  // A specific exception
  buyMoreLlamas();
} on Exception catch (e) {
  // Anything else that is an exception
  print('Unknown exception: $e');
} catch (e) {
  // No specified type, handles all
  print('Something really unknown: $e');
}
```

As the preceding code shows, you can use either on or `catch` or both. Use on when you need to specify the exception type. Use `catch` when your exception handler needs the exception object.

You can specify one or two parameters to `catch()`. The first is the exception that was thrown, and the second is the stack trace (a StackTrace object).

```
try {
  // ···
} on Exception catch (e) {
  print('Exception details:\n $e');
} catch (e, s) {
  print('Exception details:\n $e');
  print('Stack trace:\n $s');
}
```

To partially handle an exception, while allowing it to propagate, use the `rethrow` keyword.

```
final foo = '';

void misbehave() {
  try {
    foo = "You can't change a final variable's value.";
  } catch (e) {
    print('misbehave() partially handled ${e.runtimeType}.');
    rethrow; // Allow callers to see the exception.
  }
}

void main() {
  try {
    misbehave();
  } catch (e) {
    print('main() finished handling ${e.runtimeType}.');
  }
}
```

Finally

To ensure that some code runs whether or not an exception is thrown, use a `finally` clause. If no `catch` clause matches the exception, the exception is propagated after the `finally` clause runs:

```
try {
  breedMoreLlamas();
} finally {
  // Always clean up, even if an exception is thrown.
  cleanLlamaStalls();
}
```

The `finally` clause runs after any matching `catch` clauses:

```
try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e'); // Handle the exception first.
} finally {
  cleanLlamaStalls(); // Then clean up.
}
```

Learn more by reading the Exceptions section.

Classes

Dart is an object-oriented language with classes and mixin-based inheritance. Every object is an instance of a class, and all classes descend from Object. *Mixin-based inheritance* means that although every class (except for Object) has exactly one superclass, a class body can be reused in multiple class hierarchies.

To create an object, you can use the new keyword with a *constructor* for a class. Constructor names can be either *`ClassName`* or *`ClassName.identifier`*. For example:

```
var jsonData = JSON.decode('{"x":1, "y":2}');

// Create a Point using Point().
var p1 = new Point(2, 2);

// Create a Point using Point.fromJson().
var p2 = new Point.fromJson(jsonData);
```

Objects have *members* consisting of functions and data (*methods* and *instance variables*, respectively). When you call a method, you *invoke* it on an object: the method has access to that object's functions and data.

Use a dot (`.`) to refer to an instance variable or method:

```
var p = new Point(2, 2);

// Set the value of the instance variable y.
p.y = 3;

// Get the value of y.
assert(p.y == 3);

// Invoke distanceTo() on p.
num distance = p.distanceTo(new Point(4, 4));
```

Use `?.` instead of `.` to avoid an exception when the leftmost operand is null:

```
// If p is non-null, set its y value to 4.
p?.y = 4;
```

Some classes provide constant constructors. To create a compile-time constant using a constant constructor, use `const` instead of new:

```
var p = const ImmutablePoint(2, 2);
```

Constructing two identical compile-time constants results in a single, canonical instance:

```
var a = const ImmutablePoint(1, 1);
var b = const ImmutablePoint(1, 1);

assert(identical(a, b)); // They are the same instance!
```

To get an object's type at runtime, you can use Object's `runtimeType` property, which returns a Type object.

```
print('The type of a is ${a.runtimeType}');
```

The following sections discuss how to implement classes.

Instance variables

Here's how you declare instance variables:

```
class Point {
  num x; // Declare instance variable x, initially null.
  num y; // Declare y, initially null.
  num z = 0; // Declare z, initially 0.
}
```

All uninitialized instance variables have the value `null`.

All instance variables generate an implicit *getter* method. Non-final instance variables also generate an implicit *setter* method. For details, see Getters and setters.

```
class Point {
  num x;
  num y;
}

void main() {
  var point = new Point();
  point.x = 4; // Use the setter method for x.
  assert(point.x == 4); // Use the getter method for x.
  assert(point.y == null); // Values default to null.
}
```

If you initialize an instance variable where it is declared (instead of in a constructor or method), the value is set when the instance is created, which is before the constructor and its initializer list execute.

Constructors

Declare a constructor by creating a function with the same name as its class (plus, optionally, an additional identifier as described in Named constructors). The most common form of constructor, the generative constructor, creates a new instance of a class:

```
class Point {
  num x, y;

  Point(num x, num y) {
    // There's a better way to do this, stay tuned.
    this.x = x;
    this.y = y;
  }
}
```

The `this` keyword refers to the current instance.

> **Note:** Use `this` only when there is a name conflict. Otherwise, Dart style omits the `this`.

The pattern of assigning a constructor argument to an instance variable is so common, Dart has syntactic sugar to make it easy:

```
class Point {
  num x, y;

  // Syntactic sugar for setting x and y
  // before the constructor body runs.
  Point(this.x, this.y);
}
```

Default constructors

If you don't declare a constructor, a default constructor is provided for you. The default constructor has no arguments and invokes the no-argument constructor in the superclass.

Constructors aren't inherited

Subclasses don't inherit constructors from their superclass. A subclass that declares no constructors has only the default (no argument, no name) constructor.

Named constructors

Use a named constructor to implement multiple constructors for a class or to provide extra clarity:

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Named constructor
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

Remember that constructors are not inherited, which means that a superclass's named constructor is not inherited by a subclass. If you want a subclass to be created with a named constructor defined in the superclass, you must implement that constructor in the subclass.

Invoking a non-default superclass constructor

By default, a constructor in a subclass calls the superclass's unnamed, no-argument constructor. The superclass's constructor is called at the beginning of the constructor body. If an initializer list is also being used, it executes before the superclass is called. In summary, the order of execution is as follows:

1. initializer list
2. superclass's no-arg constructor
3. main class's no-arg constructor

If the superclass doesn't have an unnamed, no-argument constructor, then you must manually call one of the constructors in the superclass. Specify the superclass constructor after a colon (`:`), just before the constructor body (if any).

In the following example, the constructor for the Employee class calls the named constructor for its superclass, Person. Click the run button () to execute the code.

Because the arguments to the superclass constructor are evaluated before invoking the constructor, an argument can be an expression such as a function call:

```
class Employee extends Person {
  Employee() : super.fromJson(getDefaultData());
  // ···
}
```

Note: When using `super()` in a constructor's initialization list, put it last. For more information, see the Dart usage guide.

Warning: Arguments to the superclass constructor do not have access to `this`. For example, arguments can call static methods but not instance methods.

Initializer list

Besides invoking a superclass constructor, you can also initialize instance variables before the constructor body runs. Separate initializers with commas.

```
// Initializer list sets instance variables before
// the constructor body runs.
Point.fromJson(Map<String, num> json)
    : x = json['x'],
      y = json['y'] {
  print('In Point.fromJson(): ($x, $y)');
}
```

Warning: The right-hand side of an initializer does not have access to `this`.

Initializer lists are handy when setting up final fields. The following example initializes three final fields in an initializer list. Click the run button ( ) to execute the code.

Redirecting constructors

Sometimes a constructor's only purpose is to redirect to another constructor in the same class. A redirecting constructor's body is empty, with the constructor call appearing after a colon (:).

```
class Point {
  num x, y;

  // The main constructor for this class.
  Point(this.x, this.y);

  // Delegates to the main constructor.
  Point.alongXAxis(num x) : this(x, 0);
}
```

Constant constructors

If your class produces objects that never change, you can make these objects compile-time constants. To do this, define a `const` constructor and make sure that all instance variables are `final`.

```
class ImmutablePoint {
  static final ImmutablePoint origin =
      const ImmutablePoint(0, 0);

  final num x, y;

  const ImmutablePoint(this.x, this.y);
}
```

Factory constructors

Use the `factory` keyword when implementing a constructor that doesn't always create a new instance of its class. For example, a factory constructor might return an instance from a cache, or it might return an instance of a subtype.

The following example demonstrates a factory constructor returning objects from a cache:

```
class Logger {
  final String name;
  bool mute = false;

  // _cache is library-private, thanks to
  // the _ in front of its name.
  static final Map<String, Logger> _cache =
      <String, Logger>{};

  factory Logger(String name) {
    if (_cache.containsKey(name)) {
      return _cache[name];
    } else {
      final logger = new Logger._internal(name);
      _cache[name] = logger;
      return logger;
    }
  }

  Logger._internal(this.name);

  void log(String msg) {
    if (!mute) print(msg);
  }
}
```

Note: Factory constructors have no access to `this`.

To invoke a factory constructor, you use the new keyword:

```
var logger = new Logger('UI');
logger.log('Button clicked');
```

Methods

Methods are functions that provide behavior for an object.

Instance methods

Instance methods on objects can access instance variables and `this`. The `distanceTo()` method in the following sample is an example of an instance method:

```
import 'dart:math';

class Point {
  num x, y;

  Point(this.x, this.y);

  num distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

Getters and setters

Getters and setters are special methods that provide read and write access to an object's properties. Recall that each instance variable has an implicit getter, plus a setter if appropriate. You can create additional properties by implementing getters and setters, using the `get` and `set` keywords:

```
class Rectangle {
  num left, top, width, height;

  Rectangle(this.left, this.top, this.width, this.height);

  // Define two calculated properties: right and bottom.
  num get right => left + width;
  set right(num value) => left = value - width;
  num get bottom => top + height;
  set bottom(num value) => top = value - height;
}

void main() {
  var rect = new Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
  assert(rect.left == -8);
}
```

With getters and setters, you can start with instance variables, later wrapping them with methods, all without changing client code.

Note: Operators such as increment (++) work in the expected way, whether or not a getter is explicitly defined. To avoid any unexpected side effects, the operator calls the getter exactly once, saving its value in a temporary variable.

Abstract methods

Instance, getter, and setter methods can be abstract, defining an interface but leaving its implementation up to other classes. To make a method abstract, use a semicolon (;) instead of a method body:

```
abstract class Doer {
  // Define instance variables and methods...

  void doSomething(); // Define an abstract method.
}

class EffectiveDoer extends Doer {
  void doSomething() {
    // Provide an implementation, so the method is not abstract here
  }
}
```

Calling an abstract method results in a runtime error.

Also see Abstract classes.

Overridable operators

You can override the operators shown in the following table. For example, if you define a Vector class, you might define a + method to add two vectors.

<	+	|	[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
–	%	>>	

Here’s an example of a class that overrides the + and - operators:

```
class Vector {
  final int x, y;

  const Vector(this.x, this.y);

  /// Overrides + (a + b).
  Vector operator +(Vector v) {
    return new Vector(x + v.x, y + v.y);
  }

  /// Overrides - (a - b).
  Vector operator -(Vector v) {
    return new Vector(x - v.x, y - v.y);
  }
}

void main() {
  final v = new Vector(2, 3);
  final w = new Vector(2, 2);

  // v == (2, 3)
  assert(v.x == 2 && v.y == 3);

  // v + w == (4, 5)
  assert((v + w).x == 4 && (v + w).y == 5);

  // v - w == (0, 1)
  assert((v - w).x == 0 && (v - w).y == 1);
}
```

If you override ==, you should also override Object's `hashCode` getter. For an example of overriding == and `hashCode`, see Implementing map keys.

For more information on overriding, in general, see Extending a class.

Abstract classes

Use the `abstract` modifier to define an *abstract class*—a class that can't be instantiated. Abstract classes are useful for defining interfaces, often with some implementation. If you want your abstract class to appear to be instantiable, define a factory constructor.

Abstract classes often have abstract methods. Here's an example of declaring an abstract class that has an abstract method:

```
// This class is declared abstract and thus
// can't be instantiated.
abstract class AbstractContainer {
  // Define constructors, fields, methods...

  void updateChildren(); // Abstract method.
}
```

The following class isn't abstract, and thus can be instantiated even though it defines an abstract method:

```
class SpecializedContainer extends AbstractContainer {
  // Define constructors, fields, methods...

  void updateChildren() {
    // Provide an implementation, so the method is not abstract here
  }

  // Abstract method causes a warning but
  // doesn't prevent instantiation.
  void doSomething();
}
```

Dart 2 difference: Only abstract classes can have abstract methods. An error is reported otherwise.

Implicit interfaces

Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B's API without inheriting B's implementation, class A should implement the B interface.

A class implements one or more interfaces by declaring them in an `implements` clause and then providing the APIs required by the interfaces. For example:

```
// A person. The implicit interface contains greet().
class Person {
  // In the interface, but visible only in this library.
  final _name;

  // Not in the interface, since this is a constructor.
  Person(this._name);

  // In the interface.
  String greet(String who) => 'Hello, $who. I am $_name.';
}

// An implementation of the Person interface.
class Impostor implements Person {
  get _name => '';

  String greet(String who) => 'Hi $who. Do you know who I am?';
}

String greetBob(Person person) => person.greet('Bob');

void main() {
  print(greetBob(new Person('Kathy')));
  print(greetBob(new Impostor()));
}
```

Here’s an example of specifying that a class implements multiple interfaces:

```
class Point implements Comparable, Location {
  // ···
}
```

Extending a class

Use `extends` to create a subclass, and `super` to refer to the superclass:

```
class Television {
  void turnOn() {
    _illuminateDisplay();
    _activateIrSensor();
  }
  // ···
}

class SmartTelevision extends Television {
  void turnOn() {
    super.turnOn();
    _bootNetworkInterface();
    _initializeMemory();
    _upgradeApps();
  }
  // ···
}
```

Overriding members

Subclasses can override instance methods, getters, and setters. You can use the `@override` annotation to indicate that you are intentionally overriding a member:

```
class SmartTelevision extends Television {
  @override
  void turnOn() {
    // ···
  }
  // ···
}
```

To narrow the type of a method parameter or instance variable in code that is type safe, you can use the `covariant` keyword.

noSuchMethod()

To detect or react whenever code attempts to use a non-existent method or instance variable, you can override `noSuchMethod()`:

```
class A {
  // Unless you override noSuchMethod, using a
  // non-existent member results in a NoSuchMethodError.
  @override
  void noSuchMethod(Invocation invocation) {
    print('You tried to use a non-existent member: ' +
        '${invocation.memberName}');
  }
}
```

Dart 2 difference: In Dart 2, you **can't invoke** an unimplemented method unless **one** of the following is true:

- The receiver has the static type `dynamic`.
- The receiver has a static type that defines the unimplemented method (abstract is OK), and the dynamic type of the receiver has an implemention of `noSuchMethod()` that's different from the one in class `Object`.

For more information, see the informal nosuchMethod forwarding specification.

If you use `noSuchMethod()` to implement every possible getter, setter, and method for one or more types, then you can use the `@proxy` annotation to avoid warnings:

Dart 2 difference: The `@proxy` annotation will be removed. For more information, see site issue #442.

```
@proxy
class A {
  @override
  void noSuchMethod(Invocation invocation) {
    // ···
  }
}
```

An alternative to `@proxy`, if you know the types at compile time, is to just declare that the class implements those types.

```
class A implements SomeClass, SomeOtherClass {
  @override
  void noSuchMethod(Invocation invocation) {
    // ···
  }
}
```

For more information on annotations, see Metadata.

Enumerated types

Enumerated types, often called *enumerations* or *enums*, are a special kind of class used to represent a fixed number of constant values.

Using enums

Declare an enumerated type using the `enum` keyword:

```
enum Color { red, green, blue }
```

Each value in an enum has an `index` getter, which returns the zero-based position of the value in the enum declaration. For example, the first value has index 0, and the second value has index 1.

```
assert(Color.red.index == 0);
assert(Color.green.index == 1);
assert(Color.blue.index == 2);
```

To get a list of all of the values in the enum, use the enum's `values` constant.

```
List<Color> colors = Color.values;
assert(colors[2] == Color.blue);
```

You can use enums in switch statements, and you'll get a warning if you don't handle all of the enum's values:

```
var aColor = Color.blue;

switch (aColor) {
  case Color.red:
    print('Red as roses!');
    break;
  case Color.green:
    print('Green as grass!');
    break;
  default: // Without this, you see a WARNING.
    print(aColor); // 'Color.blue'
}
```

Enumerated types have the following limits:

- You can’t subclass, mix in, or implement an enum.
- You can’t explicitly instantiate an enum.

For more information, see the Dart Language Specification.

Adding features to a class: mixins

Mixins are a way of reusing a class’s code in multiple class hierarchies.

To use a mixin, use the `with` keyword followed by one or more mixin names. The following example shows two classes that use mixins:

```
class Musician extends Performer with Musical {
  // ···
}

class Maestro extends Person
    with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

To implement a mixin, create a class that extends Object, declares no constructors, and has no calls to `super`. For example:

```
abstract class Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```

Note: As of 1.13, two restrictions on mixins have been lifted from the Dart VM:

- Mixins allow extending from a class other than Object.
- Mixins can call `super()`.

These “super mixins” are not yet supported in dart2js and require the `--supermixin` flag in dartanalyzer.

For more information, see the article Mixins in Dart.

Class variables and methods

Use the `static` keyword to implement class-wide variables and methods.

Static variables

Static variables (class variables) are useful for class-wide state and constants:

```
class Queue {
  static const int initialCapacity = 16;
  // ···
}

void main() {
  assert(Queue.initialCapacity == 16);
}
```

Static variables aren’t initialized until they’re used.

Note: This page follows the style guide recommendation of preferring `lowerCamelCase` for constant names.

Static methods

Static methods (class methods) do not operate on an instance, and thus do not have access to `this`. For example:

```
import 'dart:math';

class Point {
  num x, y;
  Point(this.x, this.y);

  static num distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  var a = new Point(2, 2);
  var b = new Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
  print(distance);
}
```

Note: Consider using top-level functions, instead of static methods, for common or widely used utilities and functionality.

You can use static methods as compile-time constants. For example, you can pass a static method as a parameter to a constant constructor.

Generics

If you look at the API documentation for the basic array type, List, you'll see that the type is actually `List<E>`. The <…> notation marks List as a *generic* (or *parameterized*) type—a type that has formal type parameters. By convention, type variables have single-letter names, such as E, T, S, K, and V.

Why use generics?

Because types are optional in Dart 1.x, you never *have* to use generics. You might *want* to, though, for the same reason you might want to use other types in your code: types (generic or not) let you document and annotate your code, making your intent clearer.

Dart 2 note: Types won't be optional in Dart 2, but you'll still be able to omit some type annotations, thanks to type inference. For more information, see Dart 2 Updates.

For example, if you intend for a list to contain only strings, you can declare it as `List<String>` (read that as "list of string"). That way you, your fellow programmers, and your tools (such as your IDE and the Dart VM in checked mode) can detect that assigning a non-string to the list is probably a mistake. Here's an example:

```
var names = new List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
names.add(42); // Fails in checked mode (succeeds in production mode
```

Another reason for using generics is to reduce code duplication. Generics let you share a single interface and implementation between many types, while still taking advantage of checked mode and static analysis early warnings. For example, say you create an interface for caching an object:

```
abstract class ObjectCache {
  Object getByKey(String key);
  void setByKey(String key, Object value);
}
```

You discover that you want a string-specific version of this interface, so you create another interface:

```
abstract class StringCache {
  String getByKey(String key);
  void setByKey(String key, String value);
}
```

Later, you decide you want a number-specific version of this interface… You get the idea.

Generic types can save you the trouble of creating all these interfaces. Instead, you can create a single interface that takes a type parameter:

```
abstract class Cache<T> {
  T getByKey(String key);
  void setByKey(String key, T value);
}
```

In this code, T is the stand-in type. It's a placeholder that you can think of as a type that a developer will define later.

Using collection literals

List and map literals can be parameterized. Parameterized literals are just like the literals you've already seen, except that you add <*type*> (for lists) or <*keyType, valueType*> (for maps) before the opening bracket. You might use parameterized literals when you want type warnings in checked mode. Here is example of using typed literals:

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

Using parameterized types with constructors

To specify one or more types when using a constructor, put the types in angle brackets (`<...>`) just after the class name. For example:

```
var names = new List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
var nameSet = new Set<String>.from(names);
```

The following code creates a map that has integer keys and values of type View:

```
var views = new Map<int, View>();
```

Generic collections and the types they contain

Dart generic types are *reified*, which means that they carry their type information around at runtime. For example, you can test the type of a collection, even in production mode:

```
var names = new List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

However, the `is` expression checks the type of the *collection* only—not of the objects inside it. In production mode, a `List<String>` might have some non-string items in it. The solution is to either check each item's type or wrap item-manipulation code in an exception handler (see Exceptions).

> **Note:** In contrast, generics in Java use *erasure*, which means that generic type parameters are removed at runtime. In Java, you can test whether an object is a List, but you can't test whether it's a `List<String>`.

Restricting the parameterized type

When implementing a generic type, you might want to limit the types of its parameters. You can do this using `extends`.

```
// T must be SomeBaseClass or one of its descendants.
class Foo<T extends SomeBaseClass> {
  // ···
}

class Extender extends SomeBaseClass {
  // ···
}

void main() {
  // It's OK to use SomeBaseClass or any of its subclasses inside <>
  var someBaseClassFoo = new Foo<SomeBaseClass>();
  var extenderFoo = new Foo<Extender>();

  // It's also OK to use no <> at all.
  var foo = new Foo();

  // Specifying any non-SomeBaseClass type results in a warning and,
  // checked mode, a runtime error.
  // var objectFoo = new Foo<Object>();
```

Using generic methods

Initially, Dart's generic support was limited to classes. A newer syntax, called *generic methods*, allows type arguments on methods and functions:

```
T first<T>(List<T> ts) {
  // Do some initial work or error checking, then...
  T tmp = ts[0];
  // Do some additional checking or processing...
  return tmp;
}
```

Here the generic type parameter on `first` (<T>) allows you to use the type argument T in several places:

- In the function's return type (T).
- In the type of an argument (`List`<T>).
- In the type of a local variable (`T tmp`).

> **Version note:** The new syntax for generic methods was introduced in SDK 1.21. If you use generic methods, specify an SDK version of 1.21 or higher.

For more information about generics, see Using Generic Methods.

Libraries and visibility

The `import` and `library` directives can help you create a modular and shareable code base. Libraries not only provide APIs, but are a unit of privacy: identifiers that start with an underscore (_) are visible only inside the library. *Every Dart app is a library*, even if it doesn't use a `library` directive.

Libraries can be distributed using packages. See Pub Package and Asset Manager for information about pub, a package manager included in the SDK.

Using libraries

Use `import` to specify how a namespace from one library is used in the scope of another library.

For example, Dart web apps generally use the dart:html library, which they can import like this:

```
import 'dart:html';
```

The only required argument to `import` is a URI specifying the library. For built-in libraries, the URI has the special `dart:` scheme. For other libraries, you can use a file system path or the `package:` scheme. The `package:` scheme specifies libraries provided by a package manager such as the pub tool. For example:

```
import 'package:test/test.dart';
```

> **Note:** *URI* stands for uniform resource identifier. *URLs* (uniform resource locators) are a common kind of URI.

Specifying a library prefix

If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if library1 and library2 both have an Element class, then you might have code like this:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;

// Uses Element from lib1.
Element element1 = new Element();

// Uses Element from lib2.
lib2.Element element2 = new lib2.Element();
```

Importing only part of a library

If you want to use only part of a library, you can selectively import the library. For example:

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Lazily loading a library

Deferred loading (also called *lazy loading*) allows an application to load a library on demand, if and when it's needed. Here are some cases when you might use deferred loading:

- To reduce an app's initial startup time.
- To perform A/B testing—trying out alternative implementations of an algorithm, for example.
- To load rarely used functionality, such as optional screens and dialogs.

To lazily load a library, you must first import it using `deferred as`.

```
import 'package:greetings/hello.dart' deferred as hello;
```

When you need the library, invoke `loadLibrary()` using the library's identifier.

```
Future greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

In the preceding code, the `await` keyword pauses execution until the library is loaded. For more information about `async` and `await`, see asynchrony support.

You can invoke `loadLibrary()` multiple times on a library without problems. The library is loaded only once.

Keep in mind the following when you use deferred loading:

- A deferred library's constants aren't constants in the importing file. Remember, these constants don't exist until the deferred library is loaded.
- You can't use types from a deferred library in the importing file. Instead, consider moving interface types to a library imported by both the deferred library and the importing file.
- Dart implicitly inserts `loadLibrary()` into the namespace that you define using `deferred as namespace`. The `loadLibrary()` function returns a Future.

Implementing libraries

See Create Library Packages for advice on how to implement a library package, including:

- How to organize library source code.
- How to use the `export` directive.
- When to use the `part` directive.

Asynchrony support

Dart libraries are full of functions that return Future or Stream objects. These functions are *asynchronous*: they return after setting up a possibly time-consuming operation (such as I/O), without waiting for that operation to complete.

The `async` and `await` keywords support asynchronous programming, letting you write asynchronous code that looks similar to synchronous code.

Handling Futures

When you need the result of a completed Future, you have two options:

- Use `async` and `await`.
- Use the Future API, as described in the library tour.

Code that uses `async` and `await` is asynchronous, but it looks a lot like synchronous code. For example, here's some code that uses `await` to wait for the result of an asynchronous function:

```
await lookUpVersion();
```

To use `await`, code must be in a function marked as `async`:

```
Future checkVersion() async {
  var version = await lookUpVersion();
  // Do something with version
}
```

Use `try`, `catch`, and `finally` to handle errors and cleanup in code that uses `await`:

```
try {
  version = await lookUpVersion();
} catch (e) {
  // React to inability to look up the version
}
```

You can use `await` multiple times in an async function. For example, the following code waits three times for the results of functions:

```
var entrypoint = await findEntrypoint();
var exitCode = await runExecutable(entrypoint, args);
await flushThenExit(exitCode);
```

In `await` *`expression`*, the value of *`expression`* is usually a Future; if it isn't, then the value is automatically wrapped in a Future. This Future object indicates a promise to return an object. The value of `await` *`expression`* is that returned object. The await expression makes execution pause until that object is available.

If you get a compile-time error when using `await`, make sure `await` is in an async function. For example, to use `await` in your app's `main()` function, the body of `main()` must be marked as `async`:

```
Future main() async {
  checkVersion();
  print('In main: version is ${await lookUpVersion()}');
}
```

Declaring async functions

An *async function* is a function whose body is marked with the `async` modifier. Although an async function might perform time-consuming operations, it returns immediately—before any of its body executes.

> **Dart 2 difference**: Async functions won't return immediately. Instead, an async function will continue executing until it either suspends or completes. For more information, see the synchronous async start discussion in the Dart Language and Library Newsletter.

Adding the `async` keyword to a function makes it return a Future. For example, consider this synchronous function, which returns a String:

```
String lookUpVersion() => '1.0.0';
```

If you change it to be an async function—for example, because a future implementation will be time consuming—the returned value is a Future:

```
Future<String> lookUpVersion() async => '1.0.0';
```

Note that the function's body doesn't need to use the Future API. Dart creates the Future object if necessary.

Handling Streams

When you need to get values from a Stream, you have two options:

- Use `async` and an *asynchronous for loop* (`await for`).
- Use the Stream API, as described in the library tour.

> **Note:** Before using `await for`, be sure that it makes the code clearer and that you really do want to wait for all of the stream's results. For example, you usually should **not** use `await for` for UI event listeners, because UI frameworks send endless streams of events.

An asynchronous for loop has the following form:

```
await for (varOrType identifier in expression) {
  // Executes each time the stream emits a value.
}
```

The value of *`expression`* must have type Stream. Execution proceeds as follows:

1. Wait until the stream emits a value.
2. Execute the body of the for loop, with the variable set to that emitted value.
3. Repeat 1 and 2 until the stream is closed.

To stop listening to the stream, you can use a `break` or `return` statement, which breaks out of the for loop and unsubscribes from the stream.

If you get a compile-time error when implementing an asynchronous for loop, make sure the `await for` is in an async function. For example, to use an asynchronous for loop in your app's `main()` function, the body of `main()` must be marked as `async`:

```
Future main() async {
  // ...
  await for (var request in requestServer) {
    handleRequest(request);
  }
  // ...
}
```

For more information about asynchronous programming, in general, see the dart:async section of the library tour. Also see the articles Dart Language Asynchrony Support: Phase 1 and Dart Language Asynchrony Support: Phase 2, and the Dart language specification.

Generators

When you need to lazily produce a sequence of values, consider using a *generator function*. Dart has built-in support for two kinds of generator functions:

- **Synchronous** generator: Returns an **Iterable** object.
- **Asynchronous** generator: Returns a **Stream** object.

To implement a **synchronous** generator function, mark the function body as `sync*`, and use `yield` statements to deliver values:

```
Iterable<int> naturalsTo(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}
```

To implement an **asynchronous** generator function, mark the function body as `async*`, and use `yield` statements to deliver values:

```
Stream<int> asynchronousNaturalsTo(int n) async* {
  int k = 0;
  while (k < n) yield k++;
}
```

If your generator is recursive, you can improve its performance by using `yield*`:

```
Iterable<int> naturalsDownFrom(int n) sync* {
  if (n > 0) {
    yield n;
    yield* naturalsDownFrom(n - 1);
  }
}
```

For more information about generators, see the article Dart Language Asynchrony Support: Phase 2.

Callable classes

To allow your Dart class to be called like a function, implement the `call()` method.

In the following example, the `WannabeFunction` class defines a call() function that takes three strings and concatenates them, separating each with a space, and appending an exclamation. Click the run button ( ) to execute the code.

For more information on treating classes like functions, see Emulating Functions in Dart.

Isolates

Modern web browsers, even on mobile platforms, run on multi-core CPUs. To take advantage of all those cores, developers traditionally use shared-memory threads running concurrently. However, shared-state concurrency is error prone and can lead to complicated code.

Instead of threads, all Dart code runs inside of *isolates*. Each isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate.

Typedefs

In Dart, functions are objects, just like strings and numbers are objects. A *typedef*, or *function-type alias*, gives a function type a name that you can use when declaring fields and return types. A typedef retains type information when a function type is assigned to a variable.

Consider the following code, which doesn't use a typedef:

```
class SortedCollection {
  Function compare;

  SortedCollection(int f(Object a, Object b)) {
    compare = f;
  }
}

// Initial, broken implementation.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = new SortedCollection(sort);

  // All we know is that compare is a function,
  // but what type of function?
  assert(coll.compare is Function);
}
```

Type information is lost when assigning `f` to `compare`. The type of `f` is `(Object, Object)` → `int` (where → means returns), yet the type of `compare` is Function. If we change the code to use explicit names and retain type information, both developers and tools can use that information.

```
typedef int Compare(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

// Initial, broken implementation.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = new SortedCollection(sort);
  assert(coll.compare is Function);
  assert(coll.compare is Compare);
}
```

Note: Currently, typedefs are restricted to function types. We expect this to change.

Because typedefs are simply aliases, they offer a way to check the type of any function. For example:

```
typedef int Compare<T>(T a, T b);

int sort(int a, int b) => a - b;

void main() {
  assert(sort is Compare<int>); // True!
}
```

New function type syntax: Dart 1.24 introduced a new form of function types, the generic function type alias. You might use this feature if you pass around generic methods, define fields that are function types, or define arguments with generic function types. Here's an example of using the new syntax:

```
typedef F = List<T> Function<T>(T);
```

Metadata

Use metadata to give additional information about your code. A metadata annotation begins with the character `@`, followed by either a reference to a compile-time constant (such as `deprecated`) or a call to a constant constructor.

Two annotations are available to all Dart code: `@deprecated` and `@override`. For examples of using `@override`, see Extending a class. Here's an example of using the `@deprecated` annotation:

```
class Television {
  /// _Deprecated: Use [turnOn] instead._
  @deprecated
  void activate() {
    turnOn();
  }

  /// Turns the TV's power on.
  void turnOn() {
    // ···
  }
}
```

You can define your own metadata annotations. Here's an example of defining a @todo annotation that takes two arguments:

```
library todo;

class Todo {
  final String who;
  final String what;

  const Todo(this.who, this.what);
}
```

And here's an example of using that @todo annotation:

```
import 'todo.dart';

@Todo('seth', 'make this do something')
void doSomething() {
  print('do something');
}
```

Metadata can appear before a library, class, typedef, type parameter, constructor, factory, function, field, parameter, or variable declaration and before an import or export directive. You can retrieve metadata at runtime using reflection.

Comments

Dart supports single-line comments, multi-line comments, and documentation comments.

Single-line comments

A single-line comment begins with `//`. Everything between `//` and the end of line is ignored by the Dart compiler.

```
void main() {
  // TODO: refactor into an AbstractLlamaGreetingFactory?
  print('Welcome to my Llama farm!');
}
```

Multi-line comments

A multi-line comment begins with `/*` and ends with `*/`. Everything between `/*` and `*/` is ignored by the Dart compiler (unless the comment is a documentation comment; see the next section). Multi-line comments can nest.

```
void main() {
  /*
   * This is a lot of work. Consider raising chickens.

  Llama larry = new Llama();
  larry.feed();
  larry.exercise();
  larry.clean();
   */
}
```

Documentation comments

Documentation comments are multi-line or single-line comments that begin with `///` or `/**`. Using `///` on consecutive lines has the same effect as a multi-line doc comment.

Inside a documentation comment, the Dart compiler ignores all text unless it is enclosed in brackets. Using brackets, you can refer to classes, methods, fields, top-level variables, functions, and parameters. The names in brackets are resolved in the lexical scope of the documented program element.

Here is an example of documentation comments with references to other classes and arguments:

```
/// A domesticated South American camelid (Lama glama).
///
/// Andean cultures have used llamas as meat and pack
/// animals since pre-Hispanic times.
class Llama {
  String name;

  /// Feeds your llama [Food].
  ///
  /// The typical llama eats one bale of hay per week.
  void feed(Food food) {
    // ...
  }

  /// Exercises your llama with an [activity] for
  /// [timeLimit] minutes.
  void exercise(Activity activity, int timeLimit) {
    // ...
  }
}
```

In the generated documentation, `[Food]` becomes a link to the API docs for the Food class.

To parse Dart code and generate HTML documentation, you can use the SDK's documentation generation tool. For an example of generated documentation, see the Dart API documentation. For advice on how to structure your comments, see Guidelines for Dart Doc Comments.

Summary

This page summarized the commonly used features in the Dart language. More features are being implemented, but we expect that they won't break existing code. For more information, see the Dart Language Specification and Effective Dart.

To learn more about Dart's core libraries, see A Tour of the Dart Libraries.